WHITEPAPER · TESTING

Author Once, Replay Forever

The economics of AI-built test suites, and why maintenance, not authoring, decides whether a suite survives.

CodeReviewer

A product of Techware Lab · codereviewer.cloud

CONTENTS

Inside this paper

About this paper

This paper argues that the cost of automated end-to-end testing is dominated by maintenance, not authoring, and that separating the expensive act of *writing* a test from the cheap act of *running* it changes the economics. Figures in the model are illustrative. Replace them with your own.

SECTION 01

The maintenance trap

Most teams agree that end-to-end tests are valuable. They exercise the product the way a user does: sign in, click through a flow, check that the right thing happened. Yet many suites are abandoned within a year. The reason is rarely that the tests were hard to write. It is that they became expensive to keep.

Every user interface changes. Buttons get renamed, a form gains a field, a modal replaces a page. Each change breaks some tests that were perfectly correct the day before. Someone has to notice, diagnose whether the failure is a real regression or a stale test, and fix it. That work lands on the same engineers who are shipping features, so it competes with the roadmap. And it usually loses.

The predictable outcome is a suite that goes red, stays red, and stops being trusted. Once a team learns to ignore failures, the suite has negative value: it costs money to run and hides the real regressions in noise.

> **The core claim**
>
> A test suite survives only if the ongoing cost of keeping it green stays well below the value of the regressions it catches. Authoring cost is paid once; maintenance is paid forever.

SECTION 02

Where the hours actually go

It helps to split the lifetime cost of a test into four parts. Only the first is visible when a team decides to "add tests"; the other three arrive later, spread thinly across every sprint.

Cost	When it is paid	What drives it
Authoring	Once, up front	Understanding the feature, writing steps and assertions, handling login and data
Running	Every execution	Infrastructure time; negligible per run for most teams
Maintenance	Every time the app changes	UI churn, renamed elements, new steps in a flow
Triage	Every unexplained failure	Flaky timing, shared test data, environment drift

Maintenance and triage scale with two things: how often your product changes and how many tests you have. A fast-moving product with a large suite pays the most, and those are exactly the teams that need the safety net.

This is why "we will write the tests later" so often becomes "we never wrote the tests". Engineers correctly sense that each new test is a small recurring liability, not a one-time investment.

SECTION 03

Separating authoring from running

AI changes the authoring step. An assistant that can read the code behind a feature and click through the running application can learn what the feature is supposed to do and record a real test for it. That is genuinely useful. But if every execution also required AI, cost would scale with every run, and nightly or per-change replays would become a budget line to argue about.

The better design treats AI as the author and a recording as the artefact. The test is written once, reviewed, and saved as concrete steps and checks. From then on it replays deterministically, with no AI involved, as often as you like: nightly, hourly, before every release.

Author once, replay forever

Authoring uses AI: it reads the code, explores the app, proposes a plan of positive and negative cases.

Replaying does not: saved steps run the same way every time, so running more often costs nothing extra.

Repair uses AI again only when the app has changed and a test genuinely broke.

This split puts the variable cost where the value is (new coverage and real change) and makes the thing you want to do more of, running the suite, effectively free.

SECTION 04

An illustrative twelve-month model

Illustrative: replace with your own numbers

The figures below are assumptions chosen to show the shape of the calculation. They are not measurements from any customer or industry survey.

Annual human effort = A + 12 × (N × b × r) + 52 × f

where **A** is authoring hours, **N** the number of tests, **b** the share of tests broken by app changes each month, **r** the hours of human time per broken test, and **f** the weekly hours spent triaging flaky failures.

Assumption	Hand-scripted suite	Author once, replay free
Tests (N)	120	120
Human authoring time per test	1.5 h	10 min to review the plan and result
Tests broken by app changes per month (b)	8%	8% (same product churn)
Human time per broken test (r)	45 min to diagnose and fix	10 min to review an automatic repair
Flaky-failure triage per week (f)	3 h	0.5 h (flaky tests quarantined automatically)

Year-one effort	Hand-scripted	Author once
Authoring (A)	180 h	20 h
Maintenance: 12 × N × b × r	86 h	19 h
Triage: 52 × f	156 h	26 h
Total	**422 h**	**65 h**

Two things stand out even with conservative inputs. First, maintenance and triage together outweigh authoring in the scripted case: the suite costs more to keep than to build. Second, because replays carry no marginal cost in the author-once model, raising run frequency from weekly to nightly or hourly adds no human hours and no usage cost. It only shortens the time between a regression landing and someone knowing about it.

SECTION 05

Flaky tests and self-repair

A flaky test fails sometimes without anything being wrong. One flaky test is an annoyance; a handful destroy trust in the whole suite, because every red run has to be investigated before anyone believes it.

Three practices keep a suite trustworthy:

- **Quarantine, don’t delete.** A test that flips between pass and fail is benched automatically: it keeps running on its own, but no longer fails the suite until it earns its way back.
- **Repair, then re-verify.** When a test breaks because the product changed, the fix is proposed automatically and replayed to prove it passes before it counts again. Humans review; they don’t rewrite.
- **Keep the evidence.** Screenshots, video and a step-by-step trace for every replay turn "it failed" into "here is exactly where", which is most of the triage time saved.

The goal is not zero failures. It is that every failure means something.

SECTION 06

A practical starting checklist

- Pick one high-value flow (sign-up, checkout, invoicing) rather than trying to cover everything.
- Write down the three to five outcomes that must never break in that flow, including one negative case.
- Use a dedicated test account with stable data; store its credentials encrypted, never in the test itself.
- Review the proposed test plan before authoring. This is where your domain knowledge matters most.
- Replay nightly and before every release from day one; the cost is the same.
- Measure two numbers each month: regressions caught before release, and human hours spent on the suite.
- Expand coverage flow by flow only once the first suite has stayed green and trusted for a few weeks.

> **Rule of thumb**
>
> If your team would not notice the suite being switched off for a week, it is not yet trusted. Fix trust before adding coverage.

SECTION 07

How CodeReviewer helps

CodeReviewer applies the author-once model to web apps and Android apps on real phones.

- Point it at a part of your app: it reads the code, explores the running product behind your login and proposes a plan of positive and negative cases for you to curate.
- It records real tests from that plan. Authoring uses your plan’s AI allowance; replays never do. Scheduled and on-demand runs are unlimited.
- Broken tests are repaired and re-verified when your app changes; flaky tests are quarantined so the suite stays green.
- Every replay keeps screenshots, video and a step-by-step trace, and test cases export to a spreadsheet for manual QA.
- The same suites can be replayed for performance budgets and passive security checks at no extra AI cost.

Start a 15-day free trial at **codereviewer.cloud**.

See it on your own codebase.

Start a 15-day free trial. No card needed.

codereviewer.cloud